# Test Suite Reduction Does Not Necessarily Require Executing The Program Under Test*

Hermann Felbinger
Institute of Software Technology
Graz University of Technology
Graz, Austria
Email: felbinger@ist.tugraz.at

Franz Wotawa
Institute of Software Technology
Graz University of Technology
Graz, Austria
Email: wotawa@ist.tugraz.at

Mihai Nica
AVL List
Graz, Austria
Email: mihai.nica@avl.com

***Abstract*—Removing redundancies from test-suites is an important task of software testing in order to keep a test-suite as small as possible, but not to harm the test-suite's fault detection capabilities. A straightforward algorithm for test-suite reduction would select elements of the test-suite randomly and remove them if and only if the reduced test-suite fulfills the same or similar coverage or mutation score. Such algorithms rely on the execution of the program and the repeated computation of coverage or mutation score. In this paper, we present an alternative approach that purely relies on a model learned from the original test-suite without requiring the execution of the program under test. The idea is to remove those tests that do not change the learned model. In order to evaluate the approach we carried out an experimental study showing that reductions of 60–99% are possible while still keeping coverage and mutation score almost the same.**

***Index Terms*—Machine learning; Software testing; Redundancy; Coverage; Mutation Score;**

## I. Introduction

When systems and programs evolve over time, their corresponding test-suites become bigger and bigger causing a growth of computation time needed to execute the test-suite. In addition, over time some elements of the test-suite might lead to a redundancy within the test-suite [1]. Eliminating such a redundancy can be performed based on analyzing the program and identifying test cases leading to execution traces that are covered by execution traces of other tests (e.g., [2], [3]). Alternatively, we are able to remove tests that do not change the obtained coverage or mutation score value (e.g., [4], [5], [6], [7], [8], [9], [10]). All these reduction methods have in common that they require the program under test to be repeatedly executed. Depending on the computational complexity and the underlying test-suite, such reduction can take a longer time. Hence, leading to the question whether there are faster approaches for redundancy elimination. Answering this question with yes, requires to introduce such a method, which does not rely on program execution or analysis for test-suite reduction. One idea going in this direction would be to replace the program with an appropriate model, which would require additional effort if done manually (e.g., [11]). At this point we bring in another idea. Every test-suite should at least partially capture the behavior of the program under test in a sufficient way. Hence, why not using the test-suite itself to obtain the required model?

For model extraction we make use of available machine learning techniques. In this way, we are able to automate the whole test-suite reduction approach and furthermore do not require the program to be executed. The proposed test-suite reduction approach now works as the following process: We obtain a model from the original test-suite. Then we successively select test cases from the test-suite, remove them, and learn again a model from the reduced test-suite. In case the model of the original test-suite and the new model are equivalent, the reduced test-suite should have the same capabilities as the original one and we again repeat this process. Otherwise, the test case is added again to the current test-suite. The process stops in case no test case can be removed without changing the model. The most important question of course is *RQ1: Is the proposed test-suite reduction approach able to reduce a test-suite without compromising coverage or mutation score substantially?* In order to answer this question we have to evaluate the approach. For this purpose, we introduce an instantiation of the approach that relies on decision trees as models and decision tree inference as their corresponding machine learning technique where we use the C4.5 algorithm [12] for inference. Of course there are many other machine learning techniques that can be used. We selected decision trees because of the underlying application domain of embedded systems we had in mind. There the programs often implement state machines using state transfer functions, which are more or less represented using nested conditional statements.

We further on present the test-suite reduction algorithm and provide results from the empirical evaluation of the algorithm also in comparison with traditional test-suite reduction based on coverage and mutation score. The empirical evaluation is based on 7 example programs. These examples and examples of similar size (size in lines of code) were already used to evaluate model learning, test-suite reduction, and test case generation approaches (e.g., [13], [14], [15], [16], [17]). The obtained results of our model learning based reduction approach are very promising, showing substantial reductions and almost no decline of coverage and mutation score. E.g. in [18] the authors also reduced the TCAS test-suite and obtained average reductions of 76.87% and 86.88%. But they

*This work was originally published at 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)

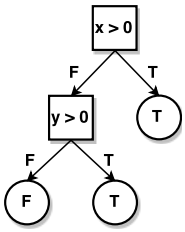


Fig. 1. Decision tree inferred from all test cases in Table I and from a subset containing only three of these test cases.

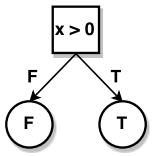


Fig. 2. Decision tree inferred from 2 of 4 test cases in Table I excluding test case #3 and #4.

also reported a huge decline of the mutation score of 55.1% and 63.95%. As shown in Section IV, we obtain for the TCAS test-suite a reduction of up to 70.87% with a max. mutation score decline of 7.32%. Compared with traditional test-suite reduction, we are able to compute the reduced test-suites in a fraction of time which is shown empirically to answer *RQ2: Is our model learning based test-suite reduction approach more efficient regarding size and execution time than a coverage and mutation score based reduction approach?*

In Example 1 we provide an introductory example of a simple test-suite TS containing 4 test cases, and show different decision trees inferred from TS and subsets of TS.

*Example 1 (Introductory example):* Table I represents an example test-suite for the Boolean expression $(x > 0) \vee (y > 0)$. A decision tree inferred from this test-suite is shown in Figure 1. The decision tree inferred from the test-suite in Table I excluding the test case with index #4 is the same decision tree as inferred from all existing test cases. The test-suite used to infer the decision tree from Figure 2 was the test-suite from Table I excluding the test cases with indexes #3 and #4.

TABLE I
TEST-SUITE FOR A BOOLEAN EXPRESSION WITH TWO INPUT VARIABLES $x$ AND $y$ OF TYPE INTEGER, BOOLEAN OUTPUT, AND # AS INDEX OF THE TEST CASES.

| # | $x$ | $y$ | $out$ |
|---|---|---|---|
| 1 | 0 | 0 | F |
| 2 | 1 | 0 | T |
| 3 | 0 | 1 | T |
| 4 | 1 | 1 | T |

The paper is organized as follows: We first discuss some preliminaries. Afterwards, we introduce the test-suite reduction algorithm. In the experimental results section we report the underlying setting of the empirical study and the obtained results. Finally, we discuss related research and conclude the paper.

## II. BASIC DEFINITIONS

In our approach we reduce test-suites *TS*, containing test cases, to subsets of these test cases. We consider a test case to be either a test vector $tc$ of *k* input values and an expected output value, e.g., $tc = (in_1, .., in_k, out)$ or a sequence $ts$ of *n* test vectors, e.g., $ts = \langle tc_1, .., tc_n \rangle$. A test case can either fail or pass. A test case passes if the expected outcomes from the vectors are equivalent to the outputs of the program under test after it was executed with the input values from the test vectors, otherwise it fails. The input types can be either discrete or numeric. Outputs are discrete. From such a test-suite we infer a decision tree.
In this work we provide a new test-suite reduction approach where we define reduction as follows:

*Definition 1 (Reduction):* Given a test-suite *TS* and a reduced test-suite *RTS* $\subseteq TS$, reduction is the difference in size of the two test-suites, given in %, and calculated as:

$$Reduction = (\frac{|TS| - |RTS|}{|TS|} 100).$$

### A. Decision tree inference

In our work we utilize the widely used C4.5 algorithm [12] to infer a decision tree $(V, E)$ from a test-suite *TS*. A decision tree is a tree $(V, E)$ having nodes $V$ and edges $E$ between nodes with the usual restrictions applying to trees. The tree has decision nodes, and leaf nodes. Each decision node represents a decision (i.e. a relational equation), e.g., $x > 0$ for numeric inputs or $x\ equals\ "open"$ for discrete inputs. A leaf node represents a classification, which is one of the discrete output values. E.g., the decision tree shown in Figure 1 has two decision nodes $x > 0$ and $y > 0$, and three leaf nodes $F$, $T$, $T$. The root node of the tree is a decision node having only outgoing edges.

### B. Coverage

In this work we assess code coverage by measuring the degree to which test cases exercise the source code of the program. There exist several different metrics to assess code coverage [19], from which we use the following in this work:

- **Statement Coverage**: To reach full statement coverage, each statement of the program code has to be executed at least once.
- **Decision Coverage**: To reach that 100% decision coverage a test-suite has to be provided s.t. each decision has a true and a false outcome at least once. In other words, each branch direction must be traversed at least once.
- **MC/DC Coverage**: To achieve 100% modified condition/decision coverage (MC/DC) requires a test-suite where each condition within a decision is shown by execution to independently and correctly affect the outcome of the decision [20].

### C. Mutation score

The mutation score is the result of mutation testing. Mutation testing [21] is a fault-based testing technique where modifications of the SUT lead to faulty versions of the SUT. These faulty versions are called mutants. A mutant is said to be a killed mutant, if a test-suite can distinguish the mutant from the original program. Mutation score is defined as:

*Definition 2 (Mutation score):* Mutation score $\mu$ is the proportion between killed mutants $r$ and the number of existing mutants $s$.

In this work we used mutants for Java source code from the categories Operator Replacement Binary an Literal Value Replacement, which are described in detail in [22].

## III. Test-Suite Reduction Approach

Our reduction approach is based on changes in the test-suite that do not cause changes in the learned model. After each reduction we learn a new model from the reduced test-suite. Whenever a reduction causes a change in the learned model, in comparison to the model we had before the reduction, we detect the change. In order to detect these changes, we have to provide a notion of equivalence for the learned models, i.e., the decision trees. In this work, we define two methods to check for equivalence, where one model is learned from the reduced test-suite *RTS*, and the other, learned from the original test-suite *TS*, serves as reference model. The first method determines syntactic equivalence of two models and the second method is based on a misclassification rate.

For simplicity, we assume a function $\rho$: *DT* $\rightarrow V$ with the universe of decision trees *DT* under consideration as input domain that returns the root node of a decision tree. E.g. the root node of the decision tree shown in Figure 1 is the decision node $x > 0$. We assume a function $\lambda$: $V \rightarrow D \cup C$ with the union of the set of decisions $D$ and the set of classifications $C$ as range that returns the content of a node. The content of a node is either a decision for decision nodes, or a classification for leaf nodes. E.g., the union of $D \cup C$ for the decision tree in Figure 1 is $\{x > 0,\, y > 0\} \cup \{F,\, T,\, T\}$. Because we infer binary decision trees the answer of a decision, e.g., whether $x > 0$ or not, is represented by a label of an outgoing edge from the decision node that can be accessed via the $\gamma$: $E \rightarrow \{$*True*, *False*$\}$ function. E.g. the outgoing edges of the root node from the decision tree in Figure 1 are labeled with $T$ and $F$ (short for *True* and *False*).

### A. Syntactic Equivalence

Two decision trees are syntactically equivalent if and only if each node in a decision tree has a corresponding node in the other decision tree, representing the identical content and each edge in the decision tree has a corresponding edge in the other decision tree with the same label and connecting the same pair of nodes. This form of equivalence can be represented using a function EQUAL: $V \times V \rightarrow \{$*True*, *False*$\}$, which we define recursively as follows: Given two decision trees $(V_1, E_1) \in$ *DT* and $(V_2, E_2) \in$ *DT*, and nodes $n_1, m_1 \in V_1$ and $n_2, m_2 \in V_2$, EQUAL returns *True*, if and only if:

1) $\lambda(n_1) = \lambda(n_2)$ (The content of the corresponding nodes has to be equivalent)
2) $|\{(n_1, m_1) | m_1$ is a direct successor of $n_1\}| = |\{(n_2, m_2) | m_2$ is a direct successor of $n_2\}|$ (The number of outgoing edges has to be equivalent)
3) $\forall(n_1, m_1)\exists(n_2, m_2)[m_1$ is a direct successor of $n_1 \wedge m_2$ is a direct successor of $n_2 \wedge \gamma(n_1, m_1) = \gamma(n_2, m_2) \wedge$ EQUAL$(m_1, m_2)]$ (The corresponding outgoing edges and the sub-decision trees have to be equivalent)

Otherwise, EQUAL returns $False$. Using this function, we define syntactic equivalence of two decision trees as follows:

*Definition 3 (Syntactic equivalence):* Given two decision trees $(V_1, E_1)$ and $(V_2, E_2)$, these decision trees are *syntactically equivalent* if and only if the function EQUAL$(\rho(V_1, E_1), \rho(V_2, E_2))$ returns $True$.

### B. Equivalence Based on a Misclassification Rate

Two decision trees, a reference decision tree $(V_1, E_1)$ inferred from a test-suite *TS*, and a decision tree $(V_2, E_2)$ inferred from a test-suite *RTS* are equivalent regarding their misclassification rate, if the following two conditions hold: First, the misclassification rate of $(V_2, E_2)$, when evaluating *TS* (therefore the reduced test cases serve as test data for the machine learning algorithm), is less than or equal to the misclassification rate of $(V_1, E_1)$. Second, for all distinct classifications which exist in $(V_1, E_1)$ an equivalent classification exists in $(V_2, E_2)$. The misclassification rate is defined as:

*Definition 4 (Misclassification rate):* The misclassification rate *MR* of a decision tree $(V, E)$ is the ratio of the number of incorrectly classified test cases *TF* $\subseteq$ *TS* to the number of all classified test cases *TS*.

$$MR(V, E) = \frac{|TF|}{|TS|} \tag{1}$$

In the remainder of this work equivalence based on the misclassification rate is named semantic equivalence. Thus we define semantic equivalence as follows:

*Definition 5 (Semantic equivalence):* A decision tree $(V_2, E_2)$ is semantically equivalent to a reference decision tree $(V_1, E_1)$ when classifying a test-suite *TS*, if the following equation holds:

$$\begin{gathered} MR(V_2, E_2) \leq MR(V_1, E_1) \wedge \\ \forall v \exists w [v \in Classifications(V_1, E_1) \wedge \\ w \in Classifications(V_2, E_2) \wedge v = w] \end{gathered}$$

The algorithm REDUCE for reducing a test-suite without executing the program under test is shown in Figure 3. As a result the algorithm yields a reduced test-suite *RTS*. REDUCE requires three inputs, i.e., a test-suite *TS*, an integer number *iterations* $> 0$, and an integer number *retries* $> 0$. *TS* is a test-suite containing all initially given test cases, *iterations* declares how often the function REDUCE is restarted, and *retries* is the maximum number of attempts, where a test case is removed from *TS* randomly, until a decision tree is inferred, which is equivalent to the decision tree inferred from *TS*. The decision trees are inferred in function `inferDecisionTree`(*TS*) using the C4.5 algorithm as explained in Section II. Equivalence of the models in REDUCE is checked either syntactically or semantically (set by configuration) as explained in Sections III-A and III-B. During execution the function REDUCE finds a subset *tRTS* of test cases from *TS* for each iteration. After each iteration of REDUCE, the size of *tRTS* is compared to the size of *RTS*. If the size of *tRTS* is smaller than the size of

```
1: function REDUCE(TS, iterations, retries)
2:     RTS = TS
3:     DT = inferDecisionTree(TS)
4:     for i = 0; i < iterations; i++ do
5:         tRTS = TS
6:         while true do
7:             found = False
8:             for j = 0; j < retries; j++ do
9:                 t = getRandomTestCase(RTS)
10:                tRTS = tRTS\t
11:                DT2 = inferDecisionTree(tRTS)
12:                if DT equals DT2 then
13:                    found = True
14:                    break
15:                else
16:                    tRTS = tRTS ∪ t
17:                end if
18:            end for
19:            if !found then
20:                break
21:            end if
22:        end while
23:        if |tRTS| < |RTS| then
24:            RTS = tRTS
25:        end if
26:    end for
27:    return RTS
28: end function
```

Fig. 3. Function to reduce a test suite *TS*.

*RTS*, *tRTS* is assigned to *RTS*. This ensures that the algorithm returns the smallest reduced test-suite which was found, but REDUCE does not guarantee to find a reduced test-suite.

## IV. EXPERIMENTAL RESULTS AND EVALUATION

In this Section we provide experimental results to answer the research questions posed in Section I.

### A. Example Programs

We evaluated our approach on seven different example programs which are:

1) **TCAS**[1]**:** An aircraft collision avoidance system for which a test-suite and mutants exist.
2) **BMI:** Categorizes the body mass index [13].
3) **Triangle:** A function that determines either one of the triangle types: equilateral, scalene, or isosceles, for valid inputs, otherwise it returns: no triangle [23].
4) **POP3:** An implementation of the state machine in [17].
5) **Car Alarm System (CAS):** An implementation of the state machine in [16].
6) **Guava UTF8**[2] **(UTF8):** A function in Google's Guava library which checks if an input sequence of bytes is a well formed UTF8 encoded input.

[1]http://sir.unl.edu/portal/bios/tcas.php

[2]https://github.com/google/guava

TABLE II
ATTRIBUTES OF THE EXAMPLE PROGRAMS.

| *Name* | *SLOC* | *classifications* | $\|TS\|$ | $s$ |
|---|---|---|---|---|
| TCAS | 100 | 3 | 1,545 | 41 |
| BMI | 19 | 5 | 1,000 | 28 |
| Triangle | 30 | 4 | 1,000 | 35 |
| POP3 | 122 | 10 | 1,000 | 167 |
| CAS | 110 | 5 | 1,000 | 167 |
| UTF8 | 56 | 2 | 1,000 | 147 |
| CC | 261 | 4 | 1,000 | 363 |

7) **Cruise Control**[3] **(CC):** Simulates a car and its cruising controller.

The attributes source lines of code (*SLOC*), number of distinct discrete output values (*classifications*), size of the test-suite *TS* ($|TS|$), and number of mutants ($s$) for the seven examples are given in Table II. The size of a test-suite is the number of test cases it contains.

### B. Reductions With Proposed Equivalence Check Methods

The test-suites of the seven example programs were reduced with a Java implementation of REDUCE. This reduction was executed using four different configurations for decision tree inference and the equivalence check. Two configurations utilized syntactical equivalence with first requiring the inference to classify a minimum of *one test case per leaf node (min 1)* and second a minimum of *two test cases per leaf node (min 2)*. The other two configurations were also first with a minimum of one test case per leaf node and second with two test cases, but with semantic equivalence. We used the values $L = \{1, 3, 5, 7, 9\}$ for *iterations* and $M = \{10, 20, 30, 40, 50\}$ for *retries* where the values for *retries* are values in % relative to $|TS|$, e.g., if $|TS| = 1000$ and $r \in M = 10\%$ then *retries* $= \frac{|TS| * r}{100} = 100$. Each of the four configurations was executed once with each of the 25 value pairs in $L \times M$.

The first results were obtained by executing REDUCE for *TS* of the TCAS example. The results for $Redcution$ are shown in Figure 4. These results show that $Reduction > 60\%$ is possible for each of the four different configurations. For the other examples we obtained similar results regarding the effect of the four different configurations, but with different results for $Reduction$.

An overview of the results for $Reduction$ from the seven examples is given in Table IV. Table IV shows the $min.$, $max.$, and $avg.$ value of $Reduction$ for each example, which were obtained by the 100 executions of REDUCE (4 configurations * 25 value pairs for *iterations* and *retries*).

### C. Test-Suite Reduction Results Using Syntactic Equivalence

In this section, to obtain a reduced test-suite *RTS* we executed REDUCE with different value pairs for *iterations* and *retries*, and syntactic equivalence. After some initial experiments we used the values $L = \{1, 2, 3, 4, 5\}$ for *iterations* and as in Section IV-B with $M = \{10, 20, 30, 40, 50\}$ for *retries*

[3]http://sir.unl.edu/portal/bios/Cruise\%20Control.php

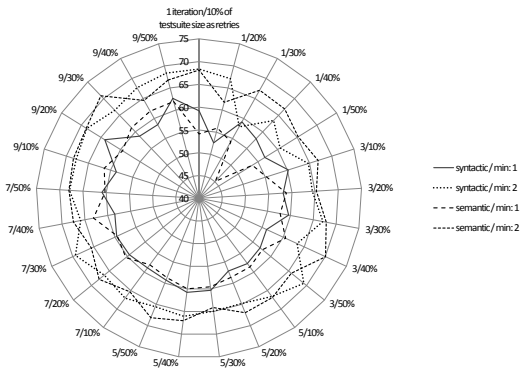


Fig. 4. *Reduction* of *TS* for the TCAS example for four different configurations and 25 executions of REDUCE for each configuration.

TABLE III
*Reductions* WITH PROPOSED EQUIVALENCE CHECK METHODS.

| | *Reduction* (%) | | |
|---|---|---|---|
| *Name* | *min.* | *max.* | *avg.* |
| TCAS | 45.11 | 70.87 | 62.49 |
| BMI | 75.10 | 82.20 | 80.08 |
| Triangle | 34.00 | 74.40 | 58.79 |
| POP3 | 95.60 | 97.50 | 96.74 |
| CAS | 98.80 | 99.70 | 99.44 |
| UTF8 | 75.80 | 91.10 | 88.27 |
| CC | 99.10 | 99.80 | 99.58 |

which are values in % relative to |*TS*|. We executed each pair in $L \times M$ (e.g., *iterations*=1, *retries*=10% of |*TS*|) 25 times.

Figure 5 shows the results for *Reduction* of the TCAS example. These reductions cut the original test-suite by around 60%. The box plots in Figure 5 are grouped by the five different values for *iterations*. These results show that for an increasing number of *iterations* the rectangles and whiskers in the plot become narrower and the median reduction value only slightly rises for an increasing number of *retries*. An example

TABLE IV
*Reductions* WITH PROPOSED EQUIVALENCE CHECK METHODS.

| | *Reduction* (%) | | | | | |
|---|---|---|---|---|---|---|
| | *syntactic* | | | *semantic* | | |
| *Name* | *min.* | *max.* | *avg.* | *min.* | *max.* | *avg.* |
| TCAS | 52.56 | 69.39 | 62.43 | 45.11 | 70.87 | 62.54 |
| BMI | 75.10 | 82.00 | 79.95 | 77.20 | 82.20 | 80.20 |
| Triangle | 34.00 | 74.40 | 58.62 | 34.70 | 74.20 | 58.96 |
| POP3 | 95.90 | 97.50 | 96.77 | 95.60 | 97.40 | 96.72 |
| CAS | 98.80 | 99.70 | 99.42 | 98.90 | 99.70 | 99.46 |
| UTF8 | 75.80 | 91.10 | 88.23 | 83.70 | 91.10 | 88.32 |
| CC | 99.20 | 99.80 | 99.61 | 99.10 | 99.80 | 99.54 |
| | *min.* 1 *per leaf node* | | | *min.* 2 *per leaf node* | | |
| *Name* | *min.* | *max.* | *avg.* | *min.* | *max.* | *avg.* |
| TCAS | 45.11 | 64.01 | 58.54 | 57.93 | 70.87 | 66.43 |
| BMI | 75.10 | 80.80 | 79.08 | 79.70 | 82.20 | 81.07 |
| Triangle | 34.00 | 74.40 | 59.56 | 38.20 | 73.20 | 58.02 |
| POP3 | 96.30 | 97.50 | 97.15 | 95.60 | 96.80 | 96.33 |
| CAS | 99.30 | 99.70 | 99.58 | 98.80 | 99.50 | 99.30 |
| UTF8 | 75.80 | 90.30 | 87.54 | 83.70 | 91.10 | 89.00 |
| CC | 99.10 | 99.80 | 99.57 | 99.30 | 99.80 | 99.59 |

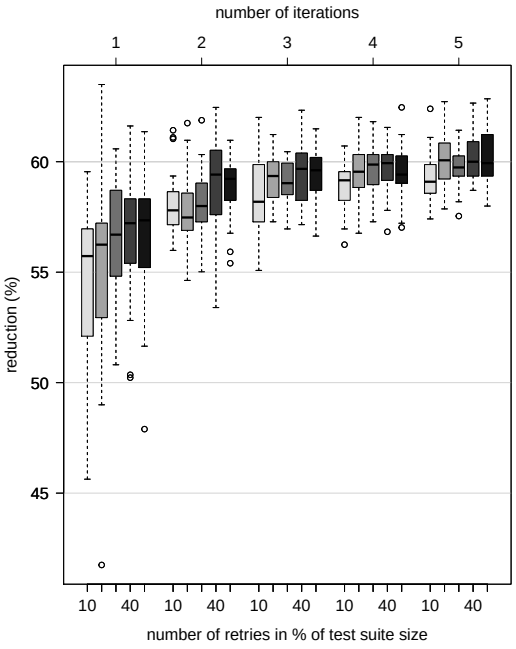


Fig. 5. *Reduction* results for the TCAS example.

test vector $tc_1 \in TS$ for the TCAS example is the vector $tc_1 = (958, 1, 1, 2597, 574, 4253, 0, 399, 400, 0, 0, 1, unresolved)$ with the last value ($unresolved$) as expected outcome and the remaining values as inputs. An example test vector $tc_2 \in TS$ for the Triangle example from which the decision trees during execution of REDUCE are inferred is the vector $tc_2 = (64, 64, 64, equilateral)$ with the last value ($equilateral$) as expected outcome and the remaining values as inputs. An example test vector $tc_3 \in TS$ for the BMI example from which the decision trees during execution of REDUCE are inferred is the vector $tc_3 = (1.717075, 183.332056, veryobese)$ with the last value ($veryobese$) as expected outcome.

An example test vector $tc_4 \in TS$ for the UTF8 example from which the decision trees during execution of REDUCE are inferred, is the vector $tc_4 = (0, ?, ?, ?, True)$ with the last value ($True$) as expected outcome and the remaining values as inputs. An input for the UTF8 example consists of up to four bytes. In $tc_4$ the input has a length of only one byte but, as the expected outcome indicates, is a valid UTF8 sequence. Decision tree inference can be used even when some input values have unknown values which are given as ? in $tc_4$. It is common to estimate missing input values for decision tree inference. For estimation either the value that is most common among the test vectors in *TS* for the input variable is used, or probabilities are estimated based on the observed frequencies of the various values for the input variable from which a value is derived.

For the POP3 example a test case is a sequence $ts = \langle tc_1, .., tc_n \rangle$ of $n$ test vectors. As input for decision tree inference we use the test vectors $tc_1, .., tc_n$ from all $ts \in TS$. An example test vector $tc_5 \in ts$ for the POP3 example,

from which the decision trees during execution of REDUCE are inferred, is the vector $tc_5 = (USER(0)$,*off*$, 1, passr)$, where the first value is the input value, the second value is the expected outcome after executing the last test vector from the current test case $ts$, the third value is a label representing the current state of the program under test, and the fourth value is the expected outcome after executing $tc_5$. The decision trees are inferred from the test vectors of all remaining test cases $ts \in$ ***RTS***, but REDUCE removes entire test cases.

For the CAS example a test case is a sequence $ts = \langle tc_1, .., tc_n \rangle$ of $n$ test vectors. An example test vector $tc_6 \in ts$, used as input for decision tree inference, is $tc_6 = (opendoor, armed, PCAlarm, activated\_flash\_alarm)$. The properties of the values in the test vectors are the same as explained for the POP3 example.

For the CC example a test case is a sequence $ts = \langle tc_1, .., tc_n \rangle$ of $n$ test vectors. As input for decision tree inference we use the test vectors $tc_1, .., tc_n$ from all $ts \in$ ***TS***. An example test vector $tc_7 \in ts$ for the CC example, from which the decision trees during execution of REDUCE are inferred, is the vector $tc_7 = (accelerator, true, 0, 1)$, where the first value is the input value, the second value is $true$, if the speed of the car is the equivalent to the speed after executing the last test vector from the current test case $ts$. The third value is represents the current state of the speed control, and the fourth value is the current state of the Cruise Control. The trend of the $Reduction$ results with increasing values for *iterations* and *retries* is for all examples similar as shown in Figure 5 for the TCAS example.

### D. Evaluation

To evaluate our results we created an adapted version of REDUCE, namely CMREDUCE, where we used coverage and mutation score for the equivalence check. CMREDUCE uses the tool CodeCover and the mutants as listed in Table II and returns the two values MC/DC coverage[4] and mutation score for the provided test-suite. Coverage and mutation score of a test-suite are equivalent to coverage and mutation score of a different test-suite if both values are equivalent.

Table V shows the differences of $Reduction$ from using REDUCE to using CMREDUCE. The execution time to obtain the results for $Reduction$ is given in Table VI. The values in Tables V and VI are the $min.$, $max.$, and $avg.$ values of the results obtained in Section IV-C and the results of executing CMREDUCE with *iterations* $= 1$ and *retries* $= 10\%$ of $|TS|$. The results in Table VI show that REDUCE is multiple times faster than CMREDUCE. Further the results in Table V show that the results of $Reduction$ differ by up to 76% for test-suites, where a test case is a single vector, but for test-suites where a test case is a sequence of vectors the maximum difference is only 3%. Knowing these initial results we answer *RQ2* as follows: The model learning based test-suite reduction approach is not more efficient regarding size than the coverage and mutation score based reduction approach, but for some examples the results are almost equivalent. Regarding execution time the model learning based reduction approach is multiple times faster. That is because for our approach it is not necessary to execute the program under test.

[4]The applied tool to measure coverage, namely CodeCover, implements the Ludewig term coverage, which subsumes MC/DC for Boolean short circuit semantics.

TABLE V
MODEL LEARNING BASED $Reduction$ RESULTS (ML BASED) VS COVERAGE+MUTATION SCORE BASED $Reduction$ (CM) RESULTS.

| | $Reduction$ (%) | | | |
|---|---|---|---|---|
| | ml based | | | cm |
| $Name$ | $min.$ | $max.$ | $avg.$ | |
| TCAS | 41.75 | 63.50 | 58.54 | 98.9 |
| BMI | 69.80 | 80.60 | 79.10 | 99.5 |
| Triangle | 22.40 | 80.00 | 59.19 | 99.3 |
| POP3 | 96.40 | 97.70 | 97.17 | 99.4 |
| CAS | 99.10 | 99.70 | 99.57 | 99.6 |
| UTF8 | 73.30 | 91.10 | 87.66 | 94.4 |
| CC | 98.00 | 99.80 | 99.50 | 99.5 |

TABLE VI
REDUCTION TIME TO OBTAIN $Reduction$ RESULTS WITH MODEL LEARNING BASED (ML BASED) AND COVERAGE+MUTATION SCORE BASED (CM) METHODS.

| | reduction time (sec) | | | |
|---|---|---|---|---|
| | ml based | | | cm |
| $Name$ | $min.$ | $max.$ | $avg.$ | |
| TCAS | 21.2 | 654.0 | 212.6 | 3692 |
| BMI | 5.2 | 81.9 | 28.5 | 1856 |
| Triangle | 9.7 | 206.8 | 69.9 | 1916 |
| POP3 | 5.1 | 58.1 | 24.0 | 5708 |
| CAS | 7.0 | 81.1 | 35.0 | 188,925 |
| UTF8 | 2.7 | 28.9 | 11.3 | 2809 |
| CC | 6.4 | 53.5 | 20.5 | 34,221 |

We further evaluated our results of $Reduction$ against a random reduction. For this random reduction we took the $avg.$ $Reduction$ of our results from Section IV-C and cut this size from the original test-suite *TS* of each example by randomly selected test cases. To get a representative result we executed this random reductions 25 times. These evaluation results are shown in Table VII. Table VII shows the $min.$, $max.$, and $avg.$ values for statement coverage, decision coverage, MC/DC coverage, and mutation score ($\mu$) for the test-suites $RTS_r$ and the test-suites $RTS_{ml}$. We obtained $RTS_r$ by random reduction and gained $RTS_{ml}$ in Section IV-C ($ml\ based$). The results in Table VII show that for the TCAS example the coverage results are not different for $RTS_r$ and $RTS_{ml}$. A difference of the mutation score for the TCAS example is shown where the mutation score for $RTS_{ml}$ is higher. For the Triangle example and the BMI example coverage and mutation score of $RTS_r$ and $RTS_{ml}$ are equivalent to the coverage and mutation score of *TS*. Mutation score of the POP3 example is for both, $RTS_r$ and $RTS_{ml}$, equivalent to the mutation score of *TS*. There are almost no differences of statement coverage, decision coverage, and MC/DC coverage for $RTS_{ml}$ to these values of *TS*. For $RTS_r$ the coverage results are considerably lower. The CAS example shows clearly that coverage and mutation score of $RTS_{ml}$ are higher than for $RTS_r$. The UTF8

example provides higher values for statement coverage for $RTS_r$ than for $RTS_{ml}$. Decision coverage, MC/DC coverage, and mutation score are higher for $RTS_{ml}$. The CC example shows that coverage and mutation score of $RTS_{ml}$ are higher than for $RTS_r$. Knowing these results we can answer *RQ1* partially. Our model learning based reduction approach does not compromise coverage and mutation score as severely as a random reduction. For examples where coverage and mutation score of the initial test-suite are at maximum or close to maximum, coverage and mutation score for the test-suites reduced by our approach, are close to coverage and mutation score of the initial test-suite. Since the results are consistently better for our model learning based approach than a random reduction, our approach might be used to reduce a test-suite.

The selected examples in this work are used to investigate whether model learning based test-suite reduction is possible in principle. Hence additional examples with more lines of code will be investigated in future work to underpin the approach. Since the structure (control flow, data flow, lines of code, etc.) of the program under test affects the approach, with more examples a possible classification can be created, where we can derive from the structure of the program under test whether the reduction approach is applicable.
In this work only two of the numerous existing methods to check whether two decision trees are equivalent were applied. Because the results are promising using these equivalence methods, additional methods will be investigated in future work. The test-suites were generated randomly for various examples. Therefore of course, as for all test-suite reduction approaches, the question, whether a test-suite which is already of minimal size for a certain test purpose, is recognized as such from the reduction approach or not, arises.

## V. Related research

Harrold et al. [6] developed a test-suite reduction technique based on a heuristic that selects a representative test-suite from the original test-suite by approximating the optimal reduced set. A representative test-suite is a potential subset of test cases from the original test-suite, which provides the same coverage. Their approach requires an association between a testing requirement and test cases, which satisfy the requirement. Since selecting a subset of test cases from the original test-suite that satisfies all testing requirements with minimal cardinality is known as the *NP-complete problem of finding the minimum cardinality hitting set*, they introduce a heuristic to approximate the minimum cardinality. They also detect redundant test cases with this approach as we do in our work, but in addition they remove obsolete test cases from a test-suite if a test requirement does not exist anymore. Lin and Huang [18] introduce a test-suite reduction technique called reduction with tie-breaking (RTB). In their paper they show the integration of their technique into two existing test-suite reduction techniques, which are the technique introduced in [6] and the technique introduced in [24]. As an extension they use, additionally to a primary coverage criterion for the test requirements, a second coverage criterion to avoid elimination of a test case, which is more likely to detect faults when more than one test case has the same importance with respect to the primary coverage criterion in both cases. In their work they provide very interesting results, which can be directly compared to the results in our work as explained in detail in Section I. Taylor et al. [25] present a multi-objective search-based technique using behavior inference as fitness metric, to reduce a test-suite. Behavior inference is a test adequacy metric first suggested in 1983 [26]. Taylor et al. infer finite state machines (FSM)s as models from execution traces of a test-suite. They compare the FSM inferred from execution traces of a reduced test-suite to the FSM inferred from execution traces of the original test-suite, and thus compare the behavior coverage of test sets that produce them. Their results show that as long as the FSMs are equivalent, assessed by a Balanced Classification Rate, the reduced test-suite retains the fault finding capability of the original test-suite. The authors demonstrate that the reduced test-suite retains all of the fault finding capability of the original test-suite by using mutation testing, which also holds for our approach. In [17] the authors provide an empirical evaluation of the correlation between mutation score and a model inference based test-suite adequacy assessment method. Briand et al. [15] describe a test-suite refinement approach that relies on the black box testing technique Category Partition and machine learning. They use categories and choices to define the functional properties of a program under test where categories are associated with choices. E.g. a category representing an inequality relation has two choices of an inequality relation which are either greater than or less than. Based on these categories they transform test cases into abstract test cases. These abstractions are tuples of choices and an expected output value or an equivalence class of expected output values. Like in our work, they use the C4.5 algorithm to learn a decision tree in [15]. But in contrast to our work where we learn a decision tree from the raw values in a test-suite, they learn decision trees from the abstractions obtained by category partitioning. Because scientific work in test-suite reduction has been done since decades, a tremendous amount of related publications exist. However, detailed overviews of test-suite reduction literature can be found in [27] and [28].

## VI. Conclusion

In this paper we introduced an algorithm for test-suite reduction, which is based on model learning. The underlying idea is, any test case that is not relevant for learning a model from a test-suite can be removed. In our implementation we utilized a decision tree inference algorithm for model learning and introduced two different notions of model equivalence. We did an empirical evaluation that is based on seven example programs and their test-suites to answer the two research questions posed in Section I. The answer for RQ1 is that coverage and mutation score for the test-suites reduced by our approach are close to coverage and mutation score of the original test-suite. Coverage and mutation score for the randomly reduced test-suites show much higher differences.

TABLE VII
RESULTS OF RANDOM REDUCED USING SAME REDUCTION SIZE AS OBTAINED BY MODEL LEARNING BASED REDUCTION VS MODEL LEARNING BASED (ML BASED) REDUCTION.

| | statement - $RTS_r$ (%) | | | statement - $RTS_{ml}$ (%) | | | decision - $RTS_r$ (%) | | | decision - $RTS_{ml}$ (%) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Name$ | $min.$ | $max.$ | $avg.$ | $min.$ | $max.$ | $avg.$ | $min.$ | $max.$ | $avg.$ | $min.$ | $max.$ | $avg.$ |
| TCAS | 97.22 | 97.22 | 97.22 | 97.22 | 97.22 | 97.22 | 91.67 | 91.67 | 91.67 | 91.67 | 91.67 | 91.67 |
| BMI | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| Triangle | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| POP3 | 84.78 | 100.00 | 93.13 | 97.80 | 100.00 | 99.99 | 90.38 | 100.00 | 94.77 | 98.08 | 100.00 | 99.99 |
| CAS | 62.30 | 95.08 | 77.77 | 93.44 | 96.72 | 96.69 | 42.86 | 83.33 | 64.19 | 80.95 | 85.71 | 85.67 |
| UTF8 | 92.85 | 100.00 | 99.14 | 85.7 | 100.00 | 96.71 | 90.00 | 100.00 | 96.20 | 90.00 | 100.00 | 97.70 |
| CC | 70.00 | 82.00 | 78.08 | 62.00 | 86.00 | 79.53 | 59.38 | 78.13 | 72.00 | 50.00 | 84.38 | 74.22 |
| | MC/DC - $RTS_r$ (%) | | | MC/DC - $RTS_{ml}$ (%) | | | $\mu$ - $RTS_r$ ([0..1]) | | | $\mu$ - $RTS_{ml}$ ([0..1]) | | |
| $Name$ | $min.$ | $max.$ | $avg.$ | $min.$ | $max.$ | $avg.$ | $min.$ | $max.$ | $avg.$ | $min.$ | $max.$ | $avg.$ |
| TCAS | 85 | 85 | 85 | 85 | 85 | 85 | 0.85 | 1.00 | 0.96 | 0.93 | 1.00 | 0.97 |
| BMI | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Triangle | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| POP3 | 94.05 | 100.00 | 96.76 | 98.80 | 100.00 | 99.99 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| CAS | 39.77 | 62.50 | 51.05 | 65.91 | 68.18 | 68.16 | 0.13 | 0.43 | 0.27 | 0.41 | 0.46 | 0.44 |
| UTF8 | 72.50 | 97.50 | 86.10 | 85.00 | 100.00 | 93.64 | 0.52 | 0.90 | 0.75 | 0.80 | 0.99 | 0.90 |
| CC | 62.50 | 81.25 | 75.13 | 50.00 | 87.50 | 77.33 | 0.18 | 0.35 | 0.30 | 0.25 | 0.38 | 0.33 |

For RQ2 the answer is that our model learning based reduction approach is multiple times faster, but not as efficient in size as a traditional approach depending on the content of a test case and the types of the inputs.

In future work, we will extend out empirical evaluation considering more examples from the application domain, i.e., automotive control software. Here the open research question is whether the program's structure, e.g., the existence of loops or recursive functions, or the use of certain expressions in the code, impacts the reduction results when using a specific machine learning algorithm. Another idea in a similar direction is the use of machine learning to obtain a measure for the quality of different test-suites. Here, we want to compare models obtained from the test-suite with the program itself. In case of a large similarity, we are able to conclude that the test-suite captures the program's behavior sufficiently.

## ACKNOWLEDGMENT

The project has received funding from the ECSEL Joint Undertaking under grant agreement No. 662192 (3Ccar). This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and the ECSEL member states.

## REFERENCES

[1] I. Pill, S. Jehan, F. Wotawa, and M. Nica, "Analyzing the reduction of test suite redundancy," in *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2015, pp. 65–65.

[2] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[3] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," in *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*. Springer-Verlag, 2007, pp. 291–305.

[4] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processessing Letters*, vol. 60, no. 3, pp. 135–141, 1996.

[5] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[6] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering Methodology*, vol. 2, no. 3, pp. 270–285, 1993.

[7] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *Proceedings of the 12th International Conference on Testing Computer Software*. ACM, 1995, pp. 111–123.

[8] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in *Proceedings of the International Conference on Software Maintenance*, 1998, pp. 34–43.

[9] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th International Conference on Software Engineering*. ACM, 1995, pp. 41–50.

[10] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, "Test set size minimization and fault detection effectiveness: a case study in a space application," in *Proceedings of the 21st International Conference on Computers, Software and Applications*, 1997, pp. 522–528.

[11] B. Korel, L. H. Tahat, and B. Vaysburg, "Model based regression test reduction using dependence analysis," in *Proceedings of the International Conference on Software Maintenance*, 2002, pp. 214–223.

[12] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[13] G. Fraser and N. Walkinshaw, "Assessing and generating test sets in terms of behavioural adequacy," *Software Testing, Verification and Reliability*, vol. 25, no. 8, pp. 749–780, 2015.

[14] P. Papadopoulos and N. Walkinshaw, "Black-box test generation from inferred models," in *Proceedings of the Fourth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, ser. RAISE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 19–24.

[15] L. C. Briand, Y. Labiche, and Z. Bawar, "Using machine learning to refine black-box test specifications and test suites," in *Proceedings of the eighth International Conference on Quality Software*, 2008, pp. 135–144.

[16] B. Aichernig, F. Lorber, and S. Tiran, "Integrating model-based testing and analysis tools via test case exchange," in *Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2012, pp. 119–126.

[17] H. Felbinger, F. Wotawa, and M. Nica, "Empirical study of correlation between mutation score and model inference based test suite adequacy assessment," in *IEEE/ACM 11th International Workshop on Automation of Software Test (AST)*, 2016.

[18] J.-W. Lin and C.-Y. Huang, "Analysis of test suite reduction with enhanced tie-breaking techniques," *Information and Software Technology*, vol. 51, no. 4, pp. 679–690, 2009.

[19] G. J. Myers and C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004.

[20] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[21] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[22] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *Proceedings of the 2012 23rd IEEE International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2012, pp. 11–20.

[23] D. Hoffman, C. Chang, G. Bazdell, B. Stevens, and K. Yoo, "Bad pairs in software testing," in *Testing – Practice and Research Techniques*, 2010, pp. 39–55.

[24] T. Chen and M. Lau, "A new heuristic for test suite reduction," *Information and Software Technology*, vol. 40, no. 56, pp. 347–354, 1998.

[25] R. Taylor, M. Hall, K. Bogdanov, and J. Derrick, "Using behaviour inference to optimise regression test sets," in *Testing Software and Systems*, 2012, pp. 184–199.

[26] E. J. Weyuker, "Assessing test data adequacy through program inference," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 4, pp. 641–655, 1983.

[27] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey." *Informatica*, vol. 35, no. 3, pp. 289–321, 2011.

[28] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verifification And Reliability*, vol. 22, no. 2, pp. 67–120, 2012.